Apunte 23 - Manejo de Excepciones y Logging Local

Pensando en el mantenimiento de nuestros Microservicios

Introducción

Este apunte aborda dos temas transversales a cualquier aplicación backend:

- **El manejo correcto de excepciones**, para garantizar la estabilidad del sistema y una comunicación clara con el usuario.
- **El uso del logging en aplicaciones Spring Boot**, para registrar, diagnosticar y auditar el comportamiento interno de nuestros microservicios.

Ambos temas forman parte del conjunto de **buenas prácticas de observabilidad** y son esenciales antes de avanzar hacia los logs centralizados y el monitoreo distribuido.

Recordemos: Conceptos básicos de Excepciones

En Java, una **excepción** es un evento que interrumpe el flujo normal de ejecución. Todas las excepciones heredan de la clase base `Throwable`, y se clasifican en:

Tipo	Descripción	Ejemplo
Checked	Deben ser manejadas o declaradas con `throws`.	`IOException`, `SQLException`
Unchecked	Heredan de `RuntimeException`. No necesitan ser declaradas.	`NullPointerException`, `IllegalArgumentException`
Errors	Errores graves del sistema (no deberían capturarse).	`OutOfMemoryError`, `StackOverflowError`

Jerarquía simplificada

```
Throwable
 ├── Error
 └── Exception
      ├── IOException
      ├── SQLException
      └── RuntimeException
           ├── NullPointerException
           ├── IllegalArgumentException
           └── ...
```

Buenas prácticas

✅ **Capturar solo lo necesario:** evita usar `catch(Exception e)` de manera indiscriminada, ya que impide distinguir el tipo de error.

Un parámetro de decisión útil es capturar una excepción solo si se puede reaccionar de alguna forma, es decir, si el código puede resolver el problema, reemplazar la excepción por otra más descriptiva o registrar un log específico. En cualquier otro caso, es preferible dejar que la excepción se propague y sea procesada por una capa superior.

Ejemplo correcto:

```
try {
    archivo.leer();
} catch (IOException e) {
    log.error("Error al leer el archivo: {}", e.getMessage());
}
```

❌ **Ejemplo incorrecto:**

```
try {
    archivo.leer();
} catch (Exception e) {
    e.printStackTrace(); // no explica el contexto ni diferencia el tipo de error
}
```

✅ **Agregar mensaje contextual** al relanzar una excepción, para facilitar el diagnóstico.

La idea es siempre aportar información acerca de la raíz del problema pensando en la persona que luego vea la consola para entender qué está ocurriendo y poder resolverlo cuando ocurra.

Ejemplo correcto:

Se aporta información acerca del problema, está en el parseo de la fecha incluso se podría aportar el formato esperado.

```
throw new DatosInvalidosException("Formato de fecha incorrecto en registro " +
registroId, e);
```

❌ **Ejemplo incorrecto:**

La palabra error o peor aún, dejarlo vacío no aporta nada a quién esté viendo el log de la excepción.

```
throw new DatosInvalidosException("Error"); // sin mensaje ni causa
```

✅ **Evitar el "swallowing" (capturar y no hacer nada):** nunca escribas bloques vacíos o silenciosos. **Ejemplo incorrecto:**

```
try {
  procesar();
}
catch (Exception e) {
  // se ignora completamente el error
}
```

Ejemplo correcto:

```
catch (Exception e) {
  log.error("Error inesperado al procesar la solicitud", e);
    throw e;
}
```

> Se captura la excepción (podríamos discutir el uso de Exception ya que siempre se busca el que catch capture la excepción específica a tratar) para luego realizar Log de esta y finalmente relanzar la execpción para que alguien más le de otro tratamiento.

✅ **No perder la traza original:** siempre incluye la excepción raíz como segundo parámetro. Esto preserva el detalle del error para los logs y depuración.

Este concepto es clave y es la razón por la que siempre una excepción acepta otra como parametro en su constructor, la idea es trabajar las excepciones como envoltorios, el más interno será la excepción que ocurrión originalmente y dio origen al tratamiento y luego iremos envolviendo esta con otras excepciones para lograr que sea procesada como esperamos.

Generalmente va a ser una capa de envoltorio pero podríamos pensar en que la capa de acceso a datos captura excepciones de JDBC o JPA y las transforma en ExcepcionesTécnicas, pero a su vez la capa de servicios captura las excepciones técnicas y las transforma en excepciones preparadas para ser retornadas al frontend.

Esta práctica se relaciona directamente con la siguiente...

✅ **Usar excepciones personalizadas** para mejorar la semántica del dominio. Estas deben comunicar el tipo de problema y su causa clara para el usuario o para la capa superior.

Excepciones personalizadas

Para comprender mejor la idea de cómo se encadenan las excepciones entre capas, podemos imaginar el siguiente esquema de envoltorio de excepciones:

```
[JDBC / JPA]
   │
   │  SQLException, PersistenceException
   ▼
[Capa de Acceso a Datos]
   │  Captura y transforma en:
   │  └── ExcepcionTecnica (incluye la causa original)
   ▼
[Capa de Servicios]
   │  Captura ExcepcionTecnica y la transforma en:
   │  └── ExcepcionDeNegocio o ExcepcionFuncional
   ▼
[Capa de Controladores]
   │  Captura ExcepcionDeNegocio y genera:
   │  └── Respuesta HTTP adecuada (404, 400, 500)
   ▼
[Frontend]
   Muestra mensaje de error amigable al usuario.
```

Este patrón de **envoltorio de excepciones** permite preservar la traza original de errores técnicos, mientras se exponen al exterior mensajes claros y controlados. Generalmente alcanza con una capa de envoltorio, pero en arquitecturas más elaboradas pueden existir varias, reflejando la responsabilidad de cada nivel del backend.

```
public class EstacionNoEncontradaException extends RuntimeException {
    public EstacionNoEncontradaException(Long id) {
        super("No se encontró la estación con ID " + id);
    }
}
```

Estas excepciones pueden lanzarse desde la capa de servicio, y luego ser capturadas globalmente.

Manejo global de excepciones con Spring

Spring Boot provee varias estrategias para interceptar y gestionar excepciones, entre ellas, pero antes de ver ejemplos conviene entender su función dentro del flujo de procesamiento de peticiones.

Cuando una excepción se lanza dentro de un método de un controlador o servicio, Spring MVC recorre una cadena de manejadores para determinar cómo responder. En este proceso:

- Si la excepción es controlada por un método anotado con `@ExceptionHandler` dentro del mismo controlador, este método se ejecuta y devuelve una respuesta específica para ese caso.
- Si la excepción no es interceptada localmente, Spring busca una clase global anotada con `@ControllerAdvice` que contenga métodos `@ExceptionHandler` aplicables. Esto permite capturar y manejar las excepciones de manera centralizada para toda la aplicación.

¿Cuándo usar cada uno?

- Usa `@ExceptionHandler` local cuando la excepción solo concierne a un controlador particular o a un conjunto pequeño de endpoints.
- Usa `@ControllerAdvice` para manejar excepciones comunes a todos los controladores, asegurando coherencia en los mensajes de error y códigos HTTP.

Centralizar la gestión de errores mejora la mantenibilidad, evita duplicación de código y garantiza que todas las respuestas sigan el mismo formato de error definido por la aplicación.

Spring Boot provee varias estrategias para interceptar y gestionar excepciones, entre ellas:

- `@ExceptionHandler` → Manejo local dentro de un controlador.
- `@ControllerAdvice` → Manejo global de toda la aplicación.

Ejemplo de `@ControllerAdvice`

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(EstacionNoEncontradaException.class)
    public ResponseEntity<ErrorResponse>
handleNotFound(EstacionNoEncontradaException ex) {
        ErrorResponse error = new ErrorResponse(LocalDateTime.now(),
                HttpStatus.NOT_FOUND.value(),
                "Recurso no encontrado",
                ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
```

```
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGeneric(Exception ex) {
        ErrorResponse error = new ErrorResponse(LocalDateTime.now(),
                HttpStatus.INTERNAL_SERVER_ERROR.value(),
                "Error interno del servidor",
                ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

```
public record ErrorResponse(LocalDateTime timestamp, int status, String error,
String message) {}
```

Logging en aplicaciones Spring Boot

Introducción al concepto de Log

Intentemos por un momento imaginar que somos desarrolladores Backend con conocimientos en microservicios y nos contratan para resolver un problema que comenzó a ocurrir en el Backend de una aplicación que estaba funcionando. En este momento pensemos nuestras ocpiones:

- Comenzar a estudiar todo el código de la aplicación para intentar inferir dónde se puede estar produciendo el problema... (realmente sería buscar una aguja en un pajar)
- O ver la aplicación funcionando para entender dónde se produce el error y atacar específicamente esa porción de código.

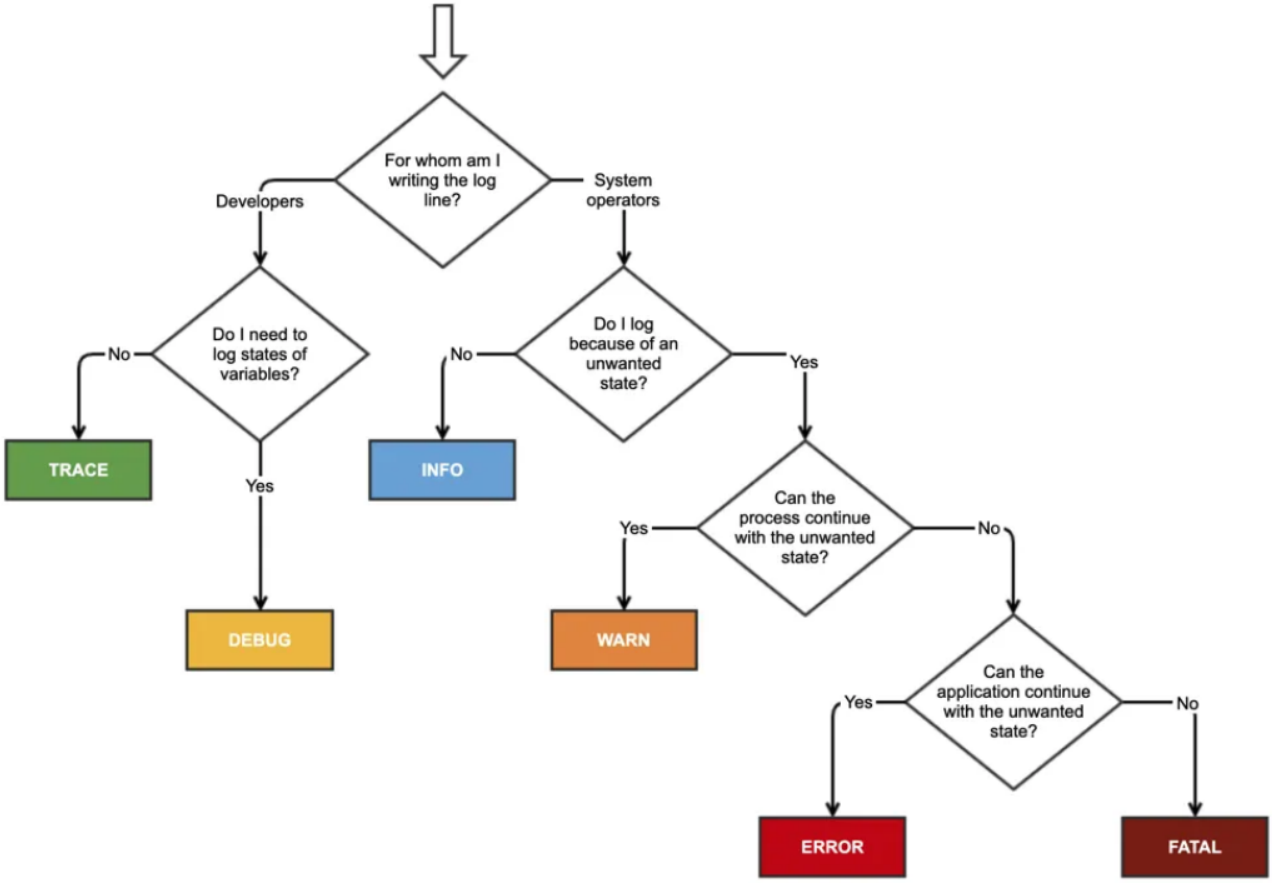
No hay dudas que el camino es el segundo, pero, en ese caso con qué nos encontramos, qué herramientas tenemos... lo único que vemos de un microservicio en funcionamiento es la terminal, y lo que podamos observar en la terminal será lo que nos marque el camino para dar con el problema y resolverlo.

Los logs son las migas de pan que el código deja en esa terminal para dar pistas de qué porcion se ejecutó correctamente y dónde se produjo el error. A continuación intentaremos documentar qué alternativas tenemos para escribir esos logs partiendo de sentencias de tipo `System.out.print(...)` y explicando por qué no son la mejor opción al uso de frameworks de logging más sofisticados.

El logging es entonces, uno de los factores clave en el mantenimiento de una Arquitectura de Microservicios. Como dijimos, vamos a escribir logging para conocer por donde pasa nuestra aplicación. Por lo general, además, vamos a escribir estos logs en diferentes niveles de log, por ejemplo: WARN, INFO, DEBUG, ERROR.

Niveles de log

Estos niveles o categorías no son arbitrarios sino que determinan uno de los elementos que debemos tener en cuenta mientras programamos al generar logs. El siguiente diagrama propone algunas preguntas para determinar el nivel del log a generar:



TRACE: Este es el nivel de registro más detallado y captura detalles minuciosos como llamadas a métodos, valores de variables y flujo de ejecución. El registro TRACE se debe utilizar para la resolución de problemas de problemas complejos.

DEBUG: Captura información sobre el comportamiento de la aplicación que puede ser útil para depurar problemas. Este nivel es adecuado para entornos de desarrollo y pruebas, donde es importante contar con información detallada sobre el comportamiento de la aplicación.

INFO: Este nivel se utiliza para capturar eventos importantes de la aplicación, como mensajes de inicio y apagado, inicio de sesión de usuarios exitosos y cambios importantes de configuración. Este nivel se debe usar en entornos de producción para capturar información que puede ayudar a identificar la causa raíz de los problemas.

WARN: Este nivel se utiliza para capturar eventos potencialmente perjudiciales, como intentos fallidos de inicio de sesión o configuraciones incorrectas. Este nivel se debe utilizar para capturar eventos que requieren atención, pero no necesariamente indican un problema crítico.

ERROR: Este nivel se utiliza para capturar errores críticos que requieren atención inmediata, como bloqueos de la aplicación o corrupción de datos.

Estos diferentes niveles de log nos van a permitir, diferente granularidad a la hora de trazar cualquier error en nuestro sistema. Si tenemos activado el nivel de traza a error, solo nos sacará aquellos mensajes en lo que se haya podido producir algún error en nuestro sistema, por ejemplo, en aquellos puntos en los que se devuelva una excepción. Por lo general se «jugará» con los diferentes niveles de log para obtener errores, o para poder hacer debug de la aplicación.

Trazabilidad en logs: con una arquitectura orientada a microservicios va a ser muy importante tener un seguimiento de las llamadas entre nuestros servicios, y poder saber quién y desde dónde ha podido ser invocado. Para ello se suele añadir un Correlation ID, el cuál es un identificador, que nos permitirá obtener todos los mensajes relacionado de una invocación a nuestro servicio.

Alertas: en un sistema en el que cada parte funciona como un engranaje con otra parte, es necesario e imprescindible que, a partir de los logs, se pueda tener un sistema de alertas.

Las alertas deben ser sistemas que nos adviertan de que algo esta fallando o va mal. Se pueden crear alertas, por ejemplo, que avisen cuando se devuelve un status code 500, lo que nos indicará que nuestro servidor da un error interno. Cuando se detecta un error, se deberá de enviar un aviso a aquellos responsables de la monitorización o propietarios del sistema para que investiguen que ha podido pasar. Para poder investigar se hará uso de las piezas vistas anteriormente.

El **logging** permite registrar eventos, errores y trazas de ejecución. Spring Boot utiliza por defecto **SLF4J + Logback**.

En resumen

Nivel	Propósito
TRACE	Detalle de ejecución, útil para depuración fina.
DEBUG	Información de diagnóstico durante desarrollo.
INFO	Eventos importantes del flujo normal.
WARN	Situaciones anómalas no críticas.
ERROR	Fallos graves o excepciones no manejadas.

Configuración básica (`application.yml`)

```yaml
logging:
  level:
    root: INFO
    org.springframework.web: DEBUG
    utnfc.isi.back: INFO
  file:
    name: logs/aplicacion.log
```

Uso de SLF4J

Antes de introducir SLF4J, vale la pena repasar la forma más básica de emitir mensajes en consola usando `System.out.println`, y por qué resulta insuficiente en aplicaciones reales.

El enfoque tradicional con System.out

Ejemplo básico:

```java
public void procesarEstacion(Long id) {
    System.out.println("Iniciando proceso para estación: " + id);
    // lógica
    System.out.println("Estación procesada correctamente");
}
```

> Este enfoque puede parecer funcional, pero presenta múltiples **limitaciones**:
>
> - No diferencia tipos de mensajes (error, advertencia, información, etc.).
> - No permite filtrar ni cambiar el nivel de detalle en tiempo de ejecución.

> - Mezcla mensajes del sistema con salida estándar, generando ruido.
> - No existe control sobre el formato ni sobre el destino del mensaje (no se guarda en archivo ni en consola estructurada).

Evolución hacia SLF4J

SLF4J (Simple Logging Facade for Java) proporciona una capa de abstracción sobre diferentes frameworks de logging como Logback o Log4j. En Spring Boot, SLF4J y Logback se incluyen por defecto, y permiten controlar fácilmente los mensajes, su formato y su destino.

Ejemplo equivalente con SLF4J:

```
@Service
public class EstacionService {
    private static final Logger log =
LoggerFactory.getLogger(EstacionService.class);

    public Estacion obtenerEstacion(Long id) {
        log.info("Iniciando búsqueda de estación con ID {}", id);
        try {
            Estacion estacion = buscarEnRepositorio(id)
                    .orElseThrow(() -> new EstacionNoEncontradaException(id));
            log.debug("Datos de estación obtenidos: {}", estacion);
            return estacion;
        } catch (EstacionNoEncontradaException e) {
            log.warn("No se encontró la estación con ID {}", id);
            throw e;
        } catch (Exception e) {
            log.error("Error inesperado al obtener estación con ID {}", id, e);
            throw e;
        }
    }
}
```

Diferencias y ventajas

Aspecto	`System.out.println`	SLF4J
Tipos de mensajes	Único (texto plano)	Múltiples niveles (`TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`)
Control en ejecución	No configurable	Configurable dinámicamente vía `application.yml` o Actuator
Destino	Consola estándar	Consola, archivo, base de datos, sistema remoto
Formato	No personalizable	Plantillas de formato y patrones definidos en Logback
Rendimiento	Sin buffering ni control	Optimizado y asincrónico (dependiendo del appender)

Configuración de niveles

Podemos establecer los niveles de log directamente en `application.yml`:

```
logging:
  level:
    root: INFO
```

```
    org.springframework.web: DEBUG
    utnfc.isi.back.logging: TRACE
```

También se pueden modificar en **tiempo de ejecución** mediante el endpoint Actuator `/actuator/loggers`, que permite subir o bajar el nivel sin reiniciar la aplicación.

Ejemplo:

```
PUT /actuator/loggers/utnfc.isi.back.logging
{
  "configuredLevel": "DEBUG"
}
```

Recomendaciones prácticas

- Utilizar `INFO` para operaciones exitosas o eventos de negocio.
- Usar `DEBUG` para información detallada de desarrollo o diagnóstico.
- Reservar `WARN` para condiciones inesperadas que no interrumpen la ejecución.
- Utilizar `ERROR` únicamente para fallas que afectan el flujo normal.
- Evitar el exceso de logs en producción: demasiados mensajes pueden degradar el rendimiento y dificultar la lectura.

Dependencias y configuración básica

Si se utiliza una versión de Spring Boot que no incluye SLF4J o Logback por defecto, puede ser necesario declarar las dependencias explícitamente en el archivo `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-logging</artifactId>
    </dependency>
</dependencies>
```

En la mayoría de los proyectos Spring Boot modernos esta dependencia ya está incorporada dentro del `spring-boot-starter-web`, pero es importante conocer su origen.

Luego, podemos realizar una configuración mínima en `application.yml`:

```
logging:
  level:
    root: INFO
    utnfc.isi.back.logging: DEBUG
  pattern:
    console: "%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n"
  file:
    name: logs/app.log
```

Esta configuración define el nivel por defecto, el formato del mensaje y el archivo donde se almacenarán los logs.

Configuración avanzada con Logback

Introducción a Logback

Logback es la implementación por defecto del sistema de logging en **Spring Boot**, sucesora directa de *Log4j* y totalmente compatible con **SLF4J**. Su diseño prioriza el rendimiento, la flexibilidad y la configuración simple mediante archivos XML.

El flujo típico es el siguiente:

```
Aplicación Java → SLF4J → Logback → Appenders (consola, archivo, etc.)
```

Cada mensaje que generamos con `log.info()` o `log.error()` pasa a través de SLF4J y finalmente es gestionado por Logback, que decide **dónde** y **cómo** escribirlo según su configuración.

Logs en archivos y concepto de Rolling Log

Cuando una aplicación se ejecuta continuamente, los logs pueden crecer indefinidamente, ocupando gran espacio en disco. Para evitarlo, Logback utiliza el concepto de **log rolling** o **rotación de archivos**. Esto implica dividir los registros en varios archivos basados en el **tiempo** (por día, hora, etc.) o en el **tamaño del archivo** (por ejemplo, cada 10 MB).

Esto permite:

- Mantener un historial de logs recientes sin saturar el disco.
- Facilitar la búsqueda y análisis de eventos por fecha.
- Borrar automáticamente los registros antiguos según la política configurada.

Máscaras de archivos de log

Logback utiliza **máscaras o patrones de nombres** para generar los archivos de log rotativos. Por ejemplo:

```
logs/app-%d{yyyy-MM-dd}.log
```

Generará archivos como:

```
logs/app-2025-11-05.log
logs/app-2025-11-06.log
```

Si el patrón se basa en tamaño, se puede usar:

```
logs/app-%i.log
```

Donde `%i` representa el número de secuencia (1, 2, 3, ...).

Ejemplo completo de configuración Logback

El archivo `logback-spring.xml` permite definir la estructura completa de logging. A continuación un ejemplo típico que combina salida por consola y archivo con rotación diaria:

```
<configuration>

    <!-- APPENDER DE CONSOLA -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%thread] %logger{36} -
%msg%n</pattern>
        </encoder>
    </appender>

    <!-- APPENDER DE ARCHIVO CON ROTACIÓN DIARIA -->
    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>logs/aplicacion.log</file>

        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>logs/aplicacion-%d{yyyy-MM-dd}.log</fileNamePattern>
            <!-- Mantener los últimos 7 días de logs -->
            <maxHistory>7</maxHistory>
        </rollingPolicy>

        <encoder>
            <pattern>%d %-5level [%thread] %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <!-- NIVEL DE LOG GENERAL -->
    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="FILE"/>
    </root>
</configuration>
```

Dependencias necesarias

Si el proyecto no incluye el *starter logging* por defecto, se deben agregar las siguientes dependencias en el `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
</dependency>

<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
</dependency>
```

Estas dependencias integran Logback con SLF4J, permitiendo definir `appenders` y políticas de rotación.

Variaciones de Rolling Policy

Tipo de rotación	Clase	Descripción

Tipo de rotación	Clase	Descripción
TimeBasedRollingPolicy	`ch.qos.logback.core.rolling.TimeBasedRollingPolicy`	Rota archivos por unidad de tiempo (día, hora, minuto).
SizeBasedTriggeringPolicy	`ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy`	Rota archivos cuando alcanzan un tamaño máximo (ej. 10MB).
FixedWindowRollingPolicy	`ch.qos.logback.core.rolling.FixedWindowRollingPolicy`	Combina índices secuenciales con tamaño fijo.

Ejemplo de rotación por tamaño:

```
<rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
    <fileNamePattern>logs/app-%i.log</fileNamePattern>
    <minIndex>1</minIndex>
    <maxIndex>5</maxIndex>
</rollingPolicy>
<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
    <maxFileSize>10MB</maxFileSize>
</triggeringPolicy>
```

Buenas prácticas con archivos de log

- Definir un tamaño o tiempo de rotación adecuado según el volumen de logs.
- Mantener una política de retención (`maxHistory`) que equilibre trazabilidad y uso de disco.
- Usar patrones de nombre claros y consistentes (`app-%d{yyyy-MM-dd}.log`).
- Configurar la carpeta `logs/` fuera del árbol de fuentes para evitar que se empaquete en el JAR.
- No escribir logs dentro de los contenedores Docker si se usan volúmenes externos (estos deben ir a `stdout` y ser recolectados por herramientas de monitoreo).

Logging por capa de aplicación

Conceptos generales

En una arquitectura multicapa, el **logging debe reflejar la responsabilidad de cada capa**. No todos los mensajes tienen la misma relevancia ni deben tener el mismo nivel de detalle. El objetivo es lograr trazabilidad sin ruido: registrar lo suficiente para entender qué ocurre, pero sin saturar los archivos de log con información irrelevante.

El enfoque recomendado es aplicar **niveles de log diferenciados** según el rol de la capa dentro de la aplicación.

Propósito de los logs en cada capa

🔷 **Capa de Controladores (Web Layer)**

- **Objetivo:** Registrar las peticiones HTTP recibidas, parámetros, encabezados relevantes y respuestas devueltas.
- **Nivel sugerido:** `INFO` para peticiones exitosas, `WARN` o `ERROR` para situaciones anómalas.
- **Buenas prácticas:**
  - Loguear la ruta (`endpoint`) y los parámetros clave.
  - No registrar información sensible (tokens, contraseñas, datos personales).
  - Utilizar *Correlation IDs* para rastrear la misma petición a través de varios servicios.

Ejemplo:

```
@GetMapping("/api/estaciones/{id}")
public ResponseEntity<Estacion> obtener(@PathVariable Long id) {
    log.info("[GET] /api/estaciones/{} - Iniciando búsqueda", id);
    Estacion estacion = service.obtenerEstacion(id);
    log.info("[GET] /api/estaciones/{} - Finalizado correctamente", id);
    return ResponseEntity.ok(estacion);
}
```

🔷 **Capa de Servicios (Service Layer)**

- **Objetivo:** Documentar la ejecución de la lógica de negocio, decisiones tomadas, validaciones y resultados intermedios.
- **Nivel sugerido:** `DEBUG` para operaciones de diagnóstico, `INFO` para eventos de negocio relevantes, `ERROR` para fallas que impidan la continuidad del proceso.
- **Buenas prácticas:**
  - Loguear antes y después de operaciones clave.
  - Capturar excepciones de negocio con contexto.
  - Evitar logs redundantes con la capa de controladores.

Ejemplo:

```
log.debug("Iniciando cálculo de tarifa para estación {}", estacionId);
if (tarifa < 0) {
    log.error("Tarifa inválida: {}", tarifa);
    throw new TarifaInvalidaException(tarifa);
}
log.info("Tarifa calculada correctamente para estación {}: {}", estacionId,
tarifa);
```

🔷 **Capa de Repositorios (Data Access Layer)**

- **Objetivo:** Registrar consultas, actualizaciones y transacciones con la base de datos.
- **Nivel sugerido:** `TRACE` o `DEBUG`.
- **Buenas prácticas:**
  - Loguear consultas o métodos de acceso (`findBy`, `save`, etc.).
  - No incluir el contenido completo de las entidades en el log (solo los identificadores o datos mínimos necesarios).
  - En entornos productivos, mantener este nivel en `WARN` o deshabilitado para evitar exceso de información.

Ejemplo:

```
log.trace("Ejecutando consulta de estaciones activas en barrio {}", barrioId);
List<Estacion> resultado = repo.findByBarrio(barrioId);
log.debug("Se recuperaron {} estaciones activas", resultado.size());
```

🔷 **Capa de Filtros e Interceptores (Infrastructure Layer)**

- **Objetivo:** Loguear la interacción global de las peticiones: tiempos de respuesta, cabeceras, sesiones, autenticación, etc.
- **Nivel sugerido:** `INFO` o `DEBUG`.
- **Buenas prácticas:**
  - Registrar la hora de inicio y fin de cada request.
  - Incluir un *request ID* o *correlation ID* para trazabilidad.
  - Evitar duplicar información ya registrada por controladores.

Ejemplo:

```
long inicio = System.currentTimeMillis();
filterChain.doFilter(request, response);
long duracion = System.currentTimeMillis() - inicio;
log.info("Petición {} procesada en {} ms", request.getRequestURI(), duracion);
```

Resumen general por capa

Capa	Nivel sugerido	Propósito principal	Ejemplo de evento típico
Controladores	INFO / WARN	Seguimiento de peticiones y respuestas HTTP	Inicio y fin de una petición REST
Servicios	DEBUG / INFO / ERROR	Lógica de negocio y validaciones	Cálculo, reglas, errores de negocio
Repositorios	TRACE / DEBUG	Acceso a datos, consultas, persistencia	Ejecución de `findBy` o `save`
Filtros / Interceptores	INFO / DEBUG	Métricas globales y trazabilidad	Tiempo de respuesta, request IDs

Anexo: Anti-patrones y errores comunes en el uso de logs

Así como un buen sistema de logging puede mejorar enormemente la mantenibilidad y trazabilidad de una aplicación, un mal uso puede tener el efecto contrario: confusión, ruido, pérdida de rendimiento y dificultad para detectar fallas reales.

Este apartado recopila los **errores más frecuentes (anti-patrones)** al registrar logs en aplicaciones Java con Spring Boot y SLF4J, junto con ejemplos y recomendaciones para evitarlos.

1. Logging redundante o duplicado

Problema: Múltiples capas registran el mismo evento, generando ruido y duplicidad.

Ejemplo incorrecto:

```
log.info("Iniciando búsqueda de estación 42");
estacionService.obtenerEstacion(42);
log.info("Estación 42 encontrada");
```

Y dentro del servicio:

```
log.info("Buscando estación 42 en base de datos");
log.info("Estación 42 encontrada correctamente");
```

Consecuencia: El mismo evento aparece registrado varias veces, complicando la lectura.

Solución: Solo la capa más cercana al usuario (por ejemplo, el controlador) debería registrar la operación de alto nivel. Las capas internas deberían limitarse a logs de diagnóstico (`DEBUG` o `TRACE`).

2. Logs sin contexto

Problema: Mensajes genéricos sin información útil.

Ejemplo incorrecto:

```
log.error("Ocurrió un error");
```

Solución: Proporcionar contexto mínimo (entidad, ID, operación). **Ejemplo correcto:**

```
log.error("Error al procesar estación con ID {}", id, e);
```

3. Logs en niveles incorrectos

Problema: Uso inadecuado de niveles que distorsiona la gravedad del mensaje.

Nivel	Uso correcto	Ejemplo de mal uso
`DEBUG`	Diagnóstico durante desarrollo	Registrar errores críticos
`INFO`	Flujo normal de eventos de negocio	Mensajes triviales en bucles
`WARN`	Condiciones anómalas no fatales	Mensajes informativos comunes
`ERROR`	Fallas graves o excepciones no manejadas	Validaciones menores

Recomendación: Ajustar el nivel según la importancia y el público del log (operadores, desarrolladores, analistas).

4. Excepciones mal registradas

Problema: No incluir la traza o registrar dos veces la misma excepción.

Ejemplo incorrecto:

```
try {
    procesar();
```

```
} catch (Exception e) {
    log.error("Error: {}", e.getMessage());
    e.printStackTrace(); // Doble registro innecesario
}
```

Ejemplo correcto:

```
catch (Exception e) {
    log.error("Error inesperado al procesar la solicitud", e);
}
```

5. Abuso de System.out o printStackTrace()

Problema: Ignorar el framework de logging y usar métodos básicos de salida.

Consecuencia: Los mensajes no respetan niveles, formatos ni configuración, y se mezclan con la salida estándar.

Solución: Utilizar siempre el logger de la clase (`log.info()`, `log.error()`, etc.).

6. Información sensible en logs

Problema: Registrar datos personales, contraseñas, tokens o claves de API.

Consecuencia: Riesgo de filtración de datos y violación de normativas de privacidad (ej. GDPR, Ley de Protección de Datos).

Ejemplo incorrecto:

```
log.info("Usuario {} autenticado con password {}", usuario, password);
```

Recomendación:

- Nunca registrar contraseñas, documentos ni tokens.
- Enmascarar valores cuando sea necesario (`****`).

---

7. Logs excesivos o poco legibles

Problema: Registrar cada paso interno del proceso sin filtrar relevancia.

Consecuencia: Archivos enormes, lentitud, dificultad de análisis.

Solución: Usar `DEBUG` o `TRACE` solo cuando sea necesario y documentar el propósito de cada log. Mantener el log `INFO` conciso.

---

8. Falta de correlación entre logs

Problema: No incluir un identificador común que permita seguir una solicitud a través de múltiples servicios.

Solución: Implementar **Correlation ID** o **Trace ID** en cada petición. Ejemplo simple:

```
UUID correlationId = UUID.randomUUID();
log.info("[{}] Procesando solicitud de estación {}", correlationId, id);
```

---

9. Logging en bucles o tareas de alta frecuencia

Problema: Generar logs dentro de operaciones repetitivas o masivas.

Ejemplo incorrecto:

```
for (Estacion e : estaciones) {
    log.info("Procesando estación {}", e.getId());
}
```

Consecuencia: Miles de líneas irrelevantes y pérdida de rendimiento.

Solución: Registrar solo eventos representativos o estadísticas resumidas.

```
log.info("Procesadas {} estaciones en {} ms", total, duracion);
```

10. Checklist de buenas prácticas

✅ Registrar eventos relevantes con contexto claro. ✅ Usar niveles de log coherentes (`DEBUG`, `INFO`, `WARN`, `ERROR`). ✅ Evitar duplicación de logs y exceso de detalle. ✅ No incluir datos sensibles. ✅ Mantener consistencia de formato y estructura. ✅ Incorporar identificadores de correlación para trazabilidad distribuida.